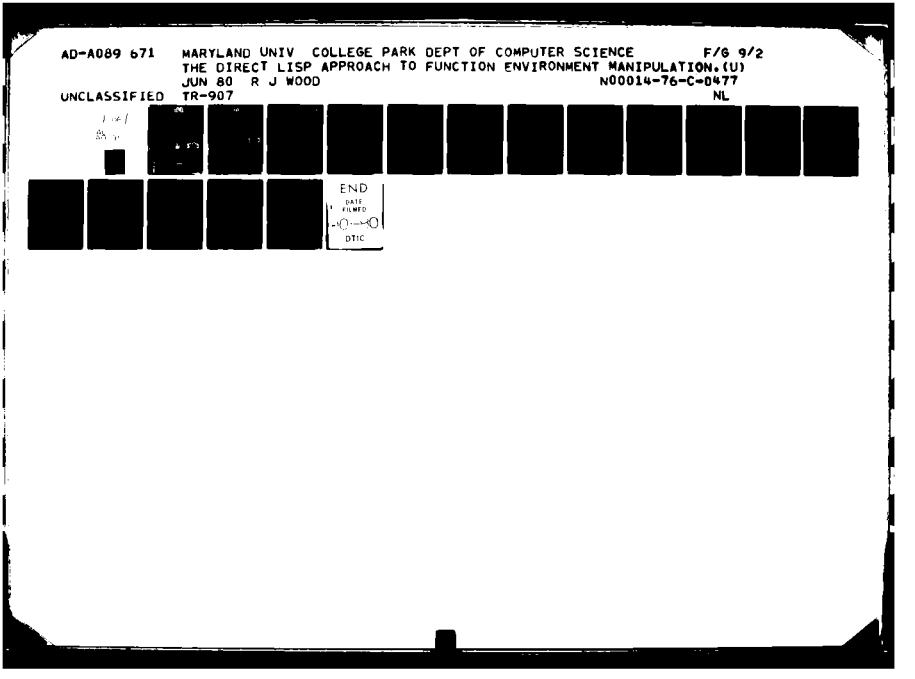
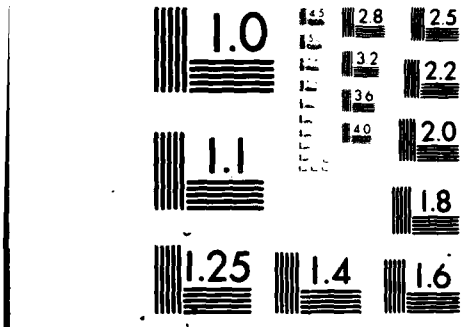
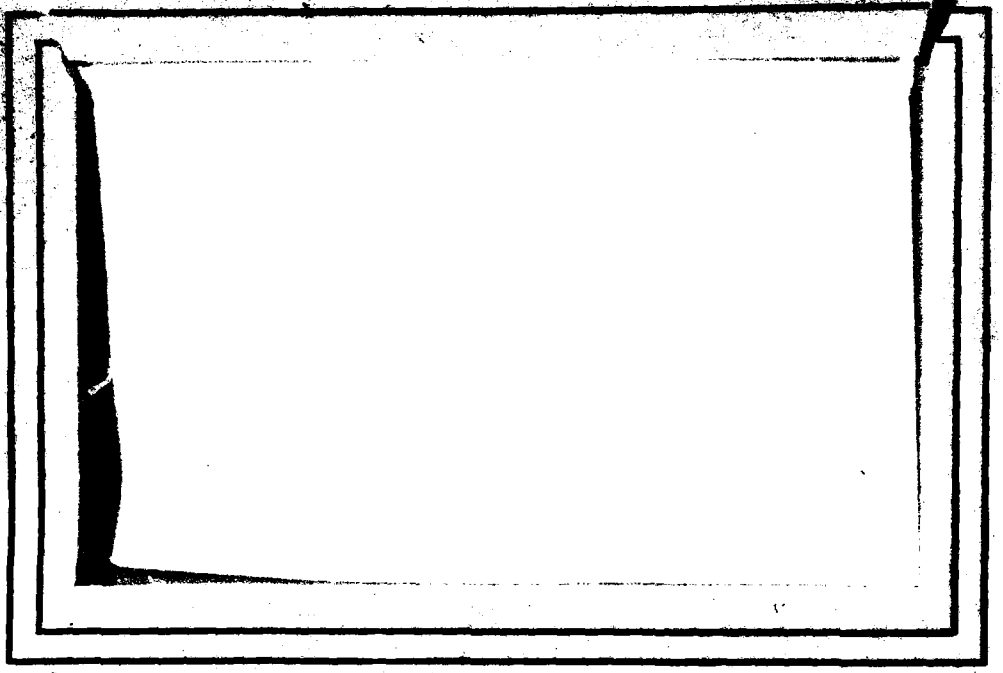

AD-A089 671 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
THE DIRECT LISP APPROACH TO FUNCTION ENVIRONMENT MANIPULATION.(U)
JUN 80 R J WOOD N00014-76-C-0477
UNCLASSIFIED TR-907 NL
END
DATE FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

AD A089671

LEVEL II



DTIC
SELECTED
SEP 30 1980
S
D
E

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

80 6 30 052

LEVEL II

8

(14) TR-907
N00014-76C-0477

(11) June 1980

(6) THE DIRECT LISP APPROACH TO FUNCTION ENVIRONMENT MANIPULATION.

(12) 17

(10) Richard J. Wood

Department of Computer Science
University of Maryland
College Park, Maryland 20742

(9) Technical Report

DTIC
ELECTE
SEP 30 1980
S D
E

Abstract: This paper reviews some of the control structures that have become popular in new extensions to LISP-based programming languages. A method for constructing these complex control structures that employs LISP features without the cost of double evaluation is developed. This technique, called the direct LISP approach, uses the LISP functions FUNCTION and EVAL to maintain and manipulate function access environments. Control environment manipulation is accomplished using continuation pointers and lambda expressions. This approach factors function execution into two distinct steps. First the functon is associated with an environment and then it is invoked. The normal LISP evaluation routines retrieve the associated environments and apply the function. This method allows the programmer flexibility to design complex control structures without expensive overhead costs. Several examples that employ the direct LISP approach are presented and discussed.

(15) The research described in this report is funded by the Office of Naval Research under grant N00014-76C-0477. Their support and encouragement are gratefully acknowledged.

409022

NTIS GRA&I
DDC TAB
Unannounced
Justification
Distribution/
Availability Codes
Dist. | Avail and/or special
A

THE DIRECT LISP APPROACH TO FUNCTION ENVIRONMENT MANIPULATION

Richard J. Wood

1. Introduction

Recent enhancements to LISP-based programming languages used primarily in Artificial Intelligence (AI) research reveal a trend toward features that support suspendable processes, coroutines, retention of environments between function activations, generator functions, and styles of computing not present in other high level languages. These new control constructs provide superior metaphors for the complex modes of processing required in many AI cognitive models and permit a user both to alter the normal control flow of a program's execution and to access variable bindings in differing environments. The implementations of these new features are based on manipulating the control and access environments associated with a function, facilities previously inaccessible to the user of a language. While these new constructs provide an extended set of operations for the user, they are often costly to implement and expensive to execute.

This paper examines several LISP programming techniques that aid a user to construct complex control structures with little overhead. Manipulation of a function's control and access environment is achieved using built-in LISP functions, requiring neither modification to the basic evaluator mechanisms nor implementation of a second-level language. The direct LISP approach provides an efficient and powerful alternative to previous methods for constructing new control paradigms.

Extensions to LISP's basic control repertoire of recursion, selection (i.e. conditionals), and iteration require modification of the strategies for retaining control and access environments. New constructs, such as suspendable processes, may retain control information past the point that it is normally accessible during function evaluation. This new retention policy, combined with the need for user accessibility to this information, requires that the normal stack-based implementation of the LISP evaluator be modified. One approach is to contruct a new mechanism to be used by the evaluator to

maintain environments, and is exemplified by Interlisp's "spaghetti stack". A second method is to represent the control and access environments as LISP data structures and to implement an evaluator for a new language containing the new constructs. This approach has been adopted by several languages (e.g. CONNIVER, Micro-PLANNER, SCHEME [Sussman75]). The former approach requires substantial modification of the LISP evaluator, while the latter suffers from slow execution caused by a double evaluation, first at the language level and then at the LISP-EVAL level.

The direct LISP approach is an efficient implementation of control constructs that employs LISP functions without the use of a second level interpreter. Modification of access environments is accomplished using FUNCTION; a feature that associates the binding environment with a user function, and EVAL; an explicit call to the LISP evaluation mechanism. Manipulation of the control environment is achieved by continuation passing; a technique of including a function's return point as an argument to the function's invocation, and LISP's equivalent treatment of program and data, permitting a function to be dynamically constructed and then executed.

This paper examines and compares the two approaches to control structure extension exemplified by CONNIVER and Interlisp. The techniques for access and control environment manipulation in LISP are presented and accompanied by several examples. The discussion concludes by assessing the relative power and efficiency of this new approach to construct new control structures.

2. Background

Interest in language features that would support "hairy control" structures originated with CONNIVER [McDermott74], a language conceived as a response to faults found in Micro-Planner [Sussman71], which itself was a partial implementation of Hewitt's PLANNER language. While not the first language to contain features such as processes and generators, CONNIVER was one of the first languages based on the philosophy that the user should have access to and be able to modify the state of the computation. This design philosophy was in part a reaction to PLANNER's hidden goal tree, which contained essentially the same information as CONNIVER's control tree, but was unavailable to the user.

CONNIVER packages a function's control environment (CLINK), access

environment (ALINK), internal variables used during evaluation, including registers and the PC (IVARS), and local variables introduced by the function (BVARS) into a user accessible data structure called a frame. This permits functions to suspend their execution and then be restarted, execute and return to another frame's caller, or use the access environment of another function to look up the bindings of non-local variables. While CONNIVER contains other new features (e.g., a context-layered data base), its greatest influence on other languages has been the use of user-accessible frames.

CONNIVER was implemented in LISP and suffers from a slow evaluation process, first at the CONNIVER language level and then at the LISP interpreter level. It was used sparingly in actual research (Fahlman's BUILD program [Fahlman73] was the major exception to this). Other languages extracted the frame philosophy and integrated this feature directly into the language design (e.g., Trigg[79] discusses a frame-based LISP interpreter).

Another approach to implementing language constructs that manipulate the evaluation environments, is to reorganize the mechanisms of the evaluator, in particular, the format of the stack. During normal function invocation and execution, values and control information are pushed and popped from the system stack. If user manipulation of environments is permitted, then sections of the stack that normally would be re-used must be preserved. This causes the stack to become tree-like in appearance (and in fact CONNIVER has a control tree constructed of CONS nodes). To implement the tree-like behavior in a true stack, link information describing the format of the tree must be included in the stack. This was the approach taken by [Bobrow73] in creating the "spaghetti stack" of Interlisp.

Both of the approaches of CONNIVER and Interlisp's spaghetti stack are based on the same philosophy; that when the user calls a function he specifies the environment(s) to use during the execution. The direct LISP approach separates the invocation of a function into two distinct operations, namely the association of an environment with a function and later the application of the function in the associated environment. While this approach is restrictive in comparison with CONNIVER, many complex CONNIVER-like control structures can nevertheless be constructed. In the following sections several examples demonstrating the power of the direct LISP approach are presented.

3. Basics: The Function FUNCTION

The LISP function FUNCTION explicitly associates an access environment with an evaluated lambda expression[1]. When the FUNCTION'ed lambda expression is applied, the access environment present when FUNCTION was executed is used to locate free variable bindings during the execution of the lambda expression. The use of the binding (or definition) environment during function execution contrasts with LISP's normal use of the activation (or runtime) environment. When the LISP interpreter encounters a FUNCTION'ed lambda expression, the current access environment is temporarily saved and the binding access environment installed *before* the lambda expression is executed. Upon completion of the function execution, the previously stored activation environment is reestablished and the result of the execution returned to the caller.

User specification of which environment, binding or activation, is to be accessed during function execution is a powerful feature not present in many higher level languages. In LISP, the FUNCTION feature solves what has been named the FUNARG problem, in which the use of functional arguments that contain references to free variables can cause unanticipated results (see [Moses70] for a complete discussion). To demonstrate the power of the FUNCTION mechanism in a role other than solving the FUNARG problem, an implementaton of the LISP function CONS is presented. This version of CONS delays the evaluation of its arguments until the function CAR or CDR is applied to the form return by CONS. The environment present during the execution of CONS must be retained until the call to CAR (or CDR). This type of delayed evaluation is due to [Friedman&Wise76] and their investigation of the semantics of CONS and Hewitt's ACTOR semantics [Hewitt77]. Figure 1 contains an encoding[2] of the function CONS and a sequence of function calls

---

[1]The lambda expression is then referred to as a FUNCTION'ed or closed lambda expression as contrasted with lambda expressions that execute in their activation environment.

[2]The examples in this paper are coded in Maryland LISP [Agre78]. Lambda expressions are represented using the following notation:

(LAMBDA <arg list> <bodies>) - Unevaluated lambda expression
[<arg list> <bodies>] - Evaluated lambda expression
[<arg list> <bodies>,E_1] - Lambda expression closed in environment E_1

where: <arg list> is the argument list of the LAMBDA expression,

that demonstrates the power of delayed evaluation using FUNCTION. The keyword FEXPR types the CONS function as being a special form that does not evaluate its arguments when it is called. CAR and CDR are regular forms that do evaluate their arguments.

```
                                                    (defun f0 ()
                                                      (f3 (f1 2 3)))

(defun cons fexpr (arg1 arg2)                       (defun f1 (a b)
  (function (lambda (sel)                             (f2 4))
              (cond ((eq sel 'car) (eval arg1))
                    ((eq sel 'cdr) (eval arg2))     (defun f2 (b)
                    (t 'error)))))                    (cons a b))

(defun car (form) (form 'car))                      (defun f3 (a)
                                                      (f4 a)
(defun cdr (form) (form 'cdr)                         (print (car a)))

                                                    (defun f4 (b)
                                                      (print (cdr b)))
```

Figure 1: Definition of CONS and the Calling Sequence

The initial call is to the function F0. The execution of the calling sequence proceeds in a straightforward manner until the call to CONS in function F2. FUNCTION establishes a pointer to the current association list, preventing it from being garbage collected, and returns a closed lambda expression containing that pointer to CONS's caller F2 (and eventually to F0 where TEMP, an explicit representation of the internal register that accumulates the result of the call to F1, is bound to the closed lambda expression). The function F3 is then invoked and passed the result of the call to F1 (i.e., TEMP). F3 calls F4 which contains a call to the function CDR (during the evaluation of the argument to PRINT). Figure 2 shows the state of the computation during the call to CDR.

When CDR is invoked it places its parameter FORM on the current association list (denoted E_s in Figure 2). When the S-expression (FORM 'CDR) is evaluated in the body of the CDR function, the old association list, E_0, becomes the current access environment (the one pointed to by E_s is stored in an internal register). The function bound to FORM is applied and its argument SEL is CONSed on the front of the current association list (and pointed at by

---

<bodies> is the list of S-expressions in the body of the LAMBDA expression.

The access environments are represented using an association list of bindings shown as CONS nodes with the CAR field pointing to the name of the variable and the CDR field pointing to the variable's value.

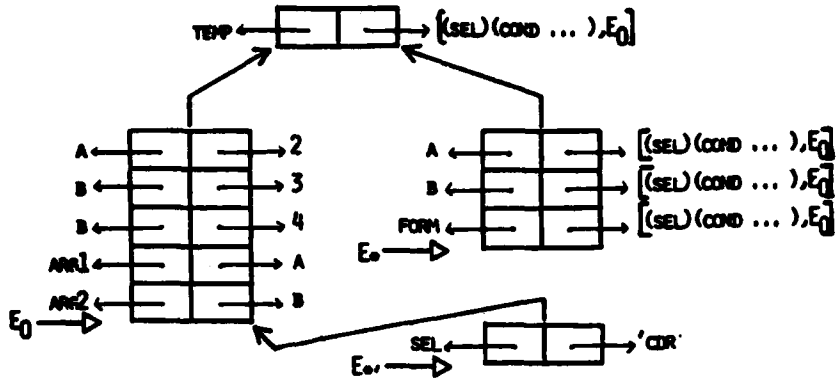


Figure 2: State of Computation during call to CDR

$E_{s'}$). Execution of the FUNCTION'ed lambda expression looks up ARG2 which is bound to B, evaluates B (executes the S-expression (EVAL ARG2) in the body of the lambda expression), and returns the value 4, which was the original second argument to CONS. When the closed lambda expression completes execution, the association list corresponding to E_s is recovered and the binding for SEL is garbage collected. The association list segment pointed at by E_0 is still retained since the binding for A (and B) in E_s (i.e. the FUNCTION'ed lambda expression) contains a pointer to E_0. During the evaluation of (PRINT (CAR A)) a similar process occurs.

The CONS example demonstrates the power of FUNCTION not only to associate an execution access environment with a function, but also to retain an access environment past the point that it would normally be garbage collected. Techniques, similar to those employed in the CONS example and based on the use of FUNCTION to manipulate environments, have been used to construct complex control paradigms. Small[80] has implemented a natural language parser organized as a series of interacting coroutines, called "word experts", that are suspended and resumed during the processing of a sentence. Each word expert routine is implemented as a closed lambda expression, that when activated restores the context of the word in the sentence before resuming execution.

4. Manipulating the Control Environment: Continuation Pointers

The CONS example demonstrates the flexibility available to a LISP programmer for designating which environment to use when accessing variables.

In a similar vein, there are instances when the normal stack-like function call and return sequence of the LISP interpreter is insufficient and needs to be modified. Coroutines and generators require that the state of the computation be retained between function calls. This includes not only the binding of variables and the access environment, but also the control path and the continuation point for the calculation. FUNCTION provides a method for manipulating access environments; control structure modifications can be accomplished using continuation pointers.

A continuation pointer is an explicit reference to the function that will be invoked when the currently executing function has completed. Normally the continuation point of a function is implicitly the next expression of the caller, and provides the standard call-return sequence that is implemented using a stack. In continuation passing, control is passed from one function to another without any function ever necessarily returning control back to its caller. The value of the executing function, which would normally be returned to the caller, is passed as an argument to the continuation point. Continuation passing factors the calculation of the return address and its placement on the stack into two distinct activities, unlike normal function invocation. In continuation passing, the calling function explicitly specifies the return point for the called function, while the called function determines when to pass control to the continuation point. The continuation point (in the form of a lambda expression) is similar to CONNIVER's CLINK of a function's frame.

Continuation passing provides an efficient implementation of a solution to the Samefringe problem, a commonly cited example of interrupted evaluation. The task is to design a function that will determine whether the fringe (set of terminal nodes) of two trees is the same, regardless of the internal structure of the tree. One solution to this problem is to calculate the complete fringe of both trees and then do an EQUAL test on the generated fringes. This method is costly in the case that the two trees differ in the first few nodes of the fringe. An alternate method calculates the first member of each fringe, compares them, and continues calculating the successive members of the fringes only while they are the same. This method has the property that it will halt at the earliest point the fringes differ.

To realize this behavior, a standard recursive tree traversal routine is

modified to suspend (i.e., momentarily halt and return a value) when it generates a member of a fringe and resume at the same point when reinvoked. Figure 3 is an encoding of a set of functions for the Samefringe problem, in which the function FRINGE has the property that it can be interrupted. The FRINGE function takes two arguments, the tree (or subtree) to be traversed, TREE, and the function to be invoked when FRINGE has finished the traversal, CP. FRINGE is called via RESUME which accepts a dotted pair whose CDR field points to a closed lambda expression. RESUME, after verifying that it has a function call, applies this expression to call FRINGE. STARTUP builds the first pair for RESUME to get the whole thing rolling[3]. The lambda expression that RESUME applies has been FUNCTION'ed causing the binding environment of the lambda expression to be reestablished before execution.

```
(defun samefringe (tree1 tree2)
  (prog ((tr1 (startup fringe tree1))
         (tr2 (startup fringe tree2)))
   loop (setq tr1 (resume tr1))
        (setq tr2 (resume tr2))
        (cond ((or (eq (get-val tr1) 'done)
                   (eq (get-val tr2) 'done)
               (return (eq (get-val tr1) (get-val tr2))))
              ((eq (get-val tr1) (get-val tr2)) (go loop))
              (t (return nil)))))

(defun fringe (tree cp)
  (cond ((atom tree) (cons tree cp))
        (t (fringe (car tree)
                   (function (lambda ()
                               (fringe (cdr tree) cp)))))))

(defun get-val (pair) (and (not (atom pair)) (car pair)))

(defun resume (pair)
  (and (not (atom pair)) (is-function-call (cdr pair))
       ((cdr pair))))
```

Figure 3: Samefringe code

A sample execution of the Samefringe functions demonstrates the power of this technique. The trees in this example are the S-expressions ((A . B) . C) and (A . (B . C)). The representation for the trees affects only the traversal routine and this technique can be extended to other representations and routines. Initially, SAMEFRINGE calls FRINGE (from RESUME) with argument TREE1. Prior to each recursive call to FRINGE a FUNCTION'ed lambda expression is formed that will invoke FRINGE on the CDR of the current value of TREE

[3]The actual value returned by STARTUP is of the form:
(nil.((fringe <tree> (lambda () 'done))))
where <tree> is either TREE1 or TREE2 depending on the exact call.

after FRINGE has completed the CAR of TREE. FRINGE recurses until the first member of the fringe, A, is located (i.e., the ATOM check succeeds). The FUNCTION'ed lambda expression, bound to CP, is CONS'ed with the newly found member of the fringe and returned to SAMEFRINGE. The association list segment constructed during the calls to FRINGE is not garbage collected because it is referenced by the closed lambda expression, which, in turn, is bound to TR1 (or TR2).

In a similar manner, FRINGE is invoked with TREE2 and locates the first member of its fringe, A. This situation is depicted in Figure 4. Note that the association list segment from the first call to FRINGE (with TREE1) has been retained and a new association list segment corresponding to the second call to FRINGE with TREE2 (denoted by $E_{\#1}$) has been added to the structure and points to the common section that contains the bindings of TREE1, TREE2, TR1, and TR2. Once again part of the current association list will be retained when FRINGE returns because it is pointed at by the FUNCTION'ed lambda expression that is bound to TR2 when FRINGE returns.

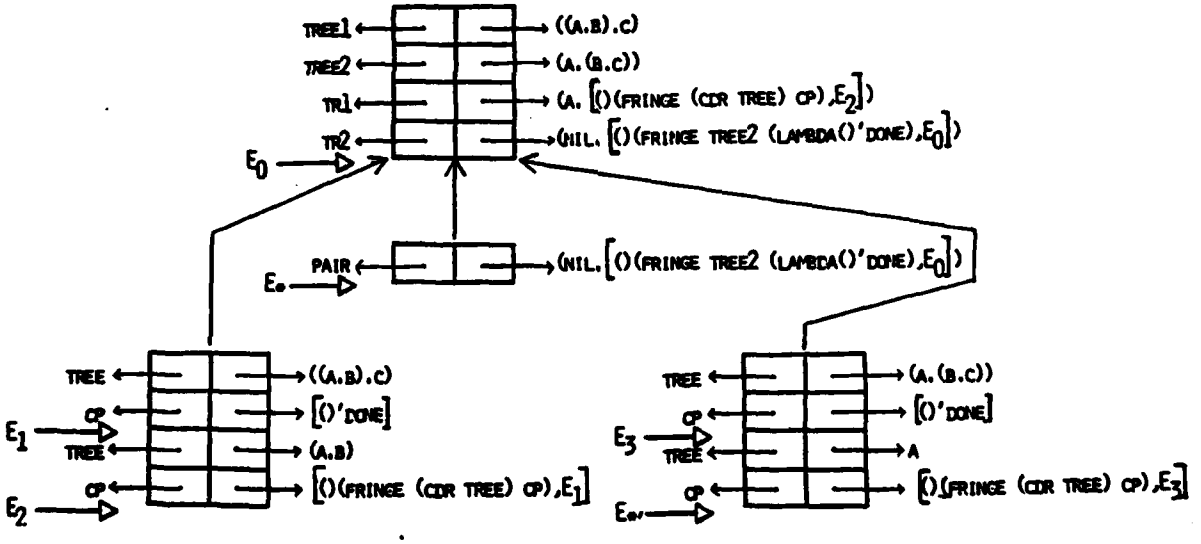


Figure 4: Calculation of the first fringe member of TREE2

The calculation continues with each member of the fringe of each tree being generated and checked against the other before calculating subsequent fringe members. When the final member of TREE1's fringe, C, is returned and bound to TR1, the last remaining pointer to E_1 is removed and that association list segment is finally garbage collected. Figure 5 shows the computation

during the calculation of TREE2's final fringe member. Upon completion of that calculation the association list segments pointed to by $E_{n'}$, E_4, E_3, and E_n will be reclaimed.

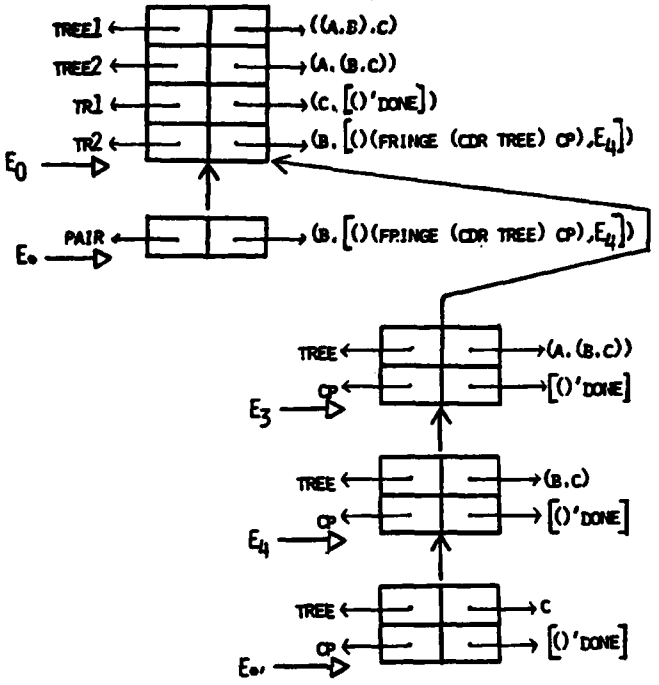


Figure 5: Calculation of the final fringe member of TREE2

Figure 6 points out an unnecessary inefficiency in this scheme for manipulating the retained environments. During the calculation of the final member of TREE2's fringe, it is not necessary to retain the E_3 and E_4 association list segments. They are retained, in fact, because the next association list segment points at them (E_4 points at E_3 and $E_{n'}$ points at E_4). The association list is constructed in a stack-like manner because of the dynamic scoping rules for non-local variable lookup. FRINGE references only the variables in its parameter list and does not invoke any functions that require non-local lookup. Thus when FRINGE is executing, it needs only the association list segment that its frame adds to the system's association list. The same mechanism that modifies the normal LISP access environment, FUNCTION, can solve this seeming inefficiency of the Samefringe problem, if FRINGE (and SAMEFRINGE, GET-VAL, and RESUME, for the same reason) is closed in its binding environment (i.e., its definition is FUNCTION'ed). This will cause the association list structure to have a higher branching factor than in

the example, but each association list segment will be retained only as long as it is needed. In the example, the E_4 segment would have been retained, without the need to retain E_3.

5. Closures

Beyond the methods for designating which environment (binding or activation) will be accessed during function execution, there are situations that require a combination of both static and dynamic scoping. In such situations the programmer might need to specify which variables will access the state of the computation at definition time, and which variables will be sensitive to the current state of the computation.

One mechanism for meeting this need is the CLOSURE function (a term introduced by the LISP machine [Greenblatt77]). CLOSURE takes a function and a list of variables to be closed (variables in the binding environment), and returns a function that when invoked will use the binding environment for looking up closed variables and the activation environment for all other free variables.

The implementation of the CLOSURE function proposed by Greenblatt[77] uses a special type of cell called an invisible pointer that the evaluator treats differently from a normal cell. This feature can be simulated by allowing EVAL to take a second argument, an association list to be used for variable look up during the EVALuation, and maintaining an explicit record of the closed variables. Figure 6 contains an encoding for the function CLOSURE.

```
(defun closure (v-lst fn)
  (eval (list 'lambda (args)
          (list 'eval (list (list 'eval fn) '(stack args))
                      (list 'append
                            (list 'quote
                                  ((lambda (a-lst)
                                     (into v-lst (lambda (x)(assoc x a-lst))))
                                   (alist)))
                            '(alist))))))

CLOSURE returns:
   [(args)(eval ((eval <fn>)(stack args))(append '(<a1.v(a1)>,..)(alist)))]
<ai.v(ai)> is a pointer to the CONS node in the association list for the
   binding of variable ai at the point of call tc CLOSURE
```

Figure 6: Closure Function

The CLOSURE function takes the argument <v-lst>, the list of variables to be closed, and constructs an association list segment of the form $\langle a_1.\text{value-of}(a_1)\rangle$ for each variable, a_1, in <v-lst>. This segment captures

the current binding of the a_i's during the execution of CLOSURE. The current association list (returned as the value of the call to the function ALIST) is appended to the end of the binding environment segment and used as a second argument to EVAL during the call to the closed function. The expression ((EVAL <fn>) (STACK <args>)) invokes the function <fn> and pushes <args> on the stack before jumping to APPLY. The values for the closed variables will be those from the binding environment, not the activation environment, even if there is another variable with the same name. (Recall that the association list is searched for the first occurence of the variable name.) CLOSURE retains the actual binding (i.e., a pointer to the CONS cell on the association list at the time of closure) in the association list segment that is constructed when it is executed, rather than a copy of the binding. This permits functions that are still active after the CLOSURE call and able to access the closed variables, to change the binding of the variable and have all the closed lambda expressions see the modification. The LISP machine CLOSURE works in a similar manner.

6. Conclusion

The direct LISP approach for manipulating access and control environments is an attractive alternative to other approaches for implementing new advanced control structures. The overhead for employing the LISP features FUNCTION and EVAL consists of storing the retained environment (which must be saved no matter what approach is used) and executing an environment switch when the function is activated. In LISP systems that use a deep binding strategy, such as Maryland LISP, environment switching involves exchanging the saved access environment pointer with the system's association list pointer, a cost of several assembly language instructions. Continuation passing is also inexpensive to employ and has been used in compilers to produce efficient code [Steele78]. The LISP features used in the direct LISP approach can be found in most LISP implementations and require no modifications to the host evaluator. Thus the direct LISP approach provides an efficient technique for extending the available control structures of LISP.

7. Acknowledgments

I would like to thank Hanan Samet, Chuck Rieger, Randy Trigg, Milt Grinberg, and Steve Small for reading drafts of this paper and making helpful

comments.

8. References

[Agre78]
Agre, P., Maryland LISP Reference Manual, Univ. of Maryland, TR-678, Jul. 1978.

[Bobrow73]
Bobrow, D.G. & Wegbreit, B., A Model and Stack Implementation of Multiple Environments, Communications of the ACM, Vol. 16,10, Oct. 1973, 591-603.

[Fahlman73]
Fahlman, S.E., A Planning System for Robot Construction Tasks, MIT A.I. Laboratory, A.I. Memo 283, 1973.

[Friedman&Wise76]
Friedman, D. & Wise, D., CONS Should Not Evaluate Its Arguments, In S. Michaelson & R. Milner, Eds., Automata, Languages, and Programming, Edinburgh: Edinburgh University Press, 1976.

[Greenblatt77]
Greenblatt, R., et al, LISP Machine Progress Report, MIT A.I. Laboratory, MIT A.I. Memo 444, Aug. 1977.

[Hewitt77]
Hewitt, C., Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol 8, 1977, pp 323-363.

[McDermott74]
McDermott, D.V. & Sussman, G.J., The CONNIVER Reference Manual, MIT A.I. Laboratory, A.I. Memo 259a, Jan. 1974.

[Moses70]
Moses, J., The Function of FUNCTION in LISP, MIT A.I. Laboratory, A.I. Memo 199, Jun. 1970.

[Small80]
Small, S.L., Word Experts for Natural Language Understanding, Univ. of Maryland, Ph.D. thesis (forthcoming).

[Steele78]
Steele, G.L., Rabbit: A Compiler for Scheme (A Study in Compiler Optimization, MIT A.I. Laboratory, A.I. Memo 474, 1978.

[Sussman71]
Sussman, G.J., Charniak, G., & Winograd, T., The Micro-Planner Reference Manual, MIT A.I. Laboratory, MIT A.I. Memo xxx, 1971.

[Sussman75]
Sussman, G.J. & Steele, G.L, SCHEME: An Interpreter for Extended Lambda Calculus, MIT A.I. Laboratory, A.I. Memo 349, Dec. 1975.

[Trigg79]
Trigg, R., A Frame Based LISP Intepreter, Univ. of Maryland, TR-777, 1979.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A089671	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Direct LISP Approach to Function Environment Manipulation		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER TR-907
7. AUTHOR(s) Richard J. Wood		8. CONTRACT OR GRANT NUMBER(s) N00014-76C-0477
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Information Systems Branch Office of Naval Research Washington, DC 20305		12. REPORT DATE June 1980
		13. NUMBER OF PAGES 13
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Programming Languages for Artificial Intelligence, LISP, Control Structures FUNARG

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This paper reviews some of the control structures that have become popular in new extensions to LISP-based programming languages. A method for constructing these complex control structures that employs LISP features without the cost of double evaluation is developed. This technique, called the direct LISP approach, uses the LISP functions FUNCTION and EVAL to maintain and manipulate function access environments. Control environment

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (con't)

manipulation is accomplished using continuation pointers and lambda expressions. This approach factors function execution into two distinct steps. First the function is associated with an environment and then it is invoked. The normal LISP evaluation routines retrieve the associated environments and apply the function. This method allows the programmer flexibility to design complex control structures without expensive overhead costs. Several examples that employ the direct LISP approach are presented and discussed.